# ViCAR: Visualizing Categories with Automated Rewriting in Coq

Bhakti Shah* William Spencer* Laura Zielinski*
Ben Caldwell Adrian Lehmann Robert Rand
University of Chicago
Chicago, USA

**Abstract.** We present ViCAR, a library for working with monoidal categories in the Coq proof assistant. ViCAR provides definitions for categorical structures that users can instantiate with their own verification projects. Upon verifying relevant coherence conditions, ViCAR gives a set of lemmas and tactics for manipulating categorical structures. We also provide a visualizer that can display any composition and tensor product of morphisms as a string diagram, showing its categorical structure. This enables graphical reasoning and automated rewriting for Coq projects with monoidal structures.

## 1 Introduction

Just as category theory provides a unifying framework for diverse concepts in mathematics, category theory can be used in formal verification as a basis for generalization. We can abstract away common structural patterns to yield reusable tactics, lemmas, and techniques for proof assistants. Many constructs that appear in program verification resemble symmetric monoidal categories. ViCAR, a library for Visualizing Categories with Automated Rewriting, takes advantage of the shared structure across symmetric monoidal instances to understand and manipulate morphisms in Coq.

ViCAR emerged from $\mathcal{V}y$ZX, an effort to verify the ZX-calculus, a graphical reasoning system for quantum programs [16]. The ZX-calculus [5] forms a symmetric monoidal category whose morphisms are the ZX-diagrams which make up the language. ViCAR generalizes the tools developed in $\mathcal{V}y$ZX to assist Coq verification for all concrete monoidal categories. ViCAR consists of three parts: typeclasses for categorical structures in Coq, a visualizer to represent morphisms using string diagrams, and a set of tactics for manipulating typeclass instances.

Other popular examples of monoidal categories include the calculus of relations and matrices. The former is the category whose objects are types and morphisms are binary relations. The latter has matrices as its morphisms between vector spaces. The similarities between these examples are initially unclear. Reframing each categorically, however, reveals their shared structure and hints at how the verification of one could help another. To explore and justify this claim, we implemented each of these instances in Coq then applied ViCAR's monoidal category framework. We found that for a well-developed project, we could easily instantiate the relevant categorical definitions. The tactics and visualization gained were valuable and helped project-specific proofs by removing proof and cognitive overhead. We discuss these examples in more detail in Section 6.

ViCAR's key contributions are as follows:

- We define symmetric monoidal categories in Coq, easily instantiable by user-created structures.

*Equal contribution

Submitted to:

© Shah, Spencer, Zielinski, Caldwell, Lehmann & Rand
This work is licensed under the
Creative Commons Attribution License.

- We present an automatic morphism visualizer. While working on a Coq proof, the visualizer will parse the monoidal structure of the current proof state, producing an image of its string diagram representation.
- We provide a set of powerful automation tactics that users can access once they have proven the necessary coherence conditions. They include `foliate`, which automatically rewrites diagrams to common useful structures; `assoc_rw`, which performs rewriting modulo associativity; and `cat_simpl`, a simplification tactic for commonly occurring patterns.

We describe ViCAR's contributions in detail, how to use them, and how they fit into $VyZX$ and other projects with categorical structure. We conclude the paper with some discussion of the next steps for ViCAR. Our work is open source and on GitHub: `https://github.com/inQWIRE/VICaR`.

## 2 Background

**Formal verification and the Coq proof assistant** Formal verification is grounded in the idea that we can mathematically prove that a computer program satisfies a specification. This guarantees that a given piece of software performs as expected on all possible inputs. Formal verification research focuses on developing new and more effective techniques for these kinds of proofs. One approach uses proof assistants, which are software (usually programming languages) that allow users to express and prove various constructs, often mathematical, in code.

Among the most widely-used proof assistants is Coq [7]. It allows for writing definitions in the style of a dependently-typed programming language and proving statements in the style of mathematical proof. One powerful Coq mechanism is custom tactics, written in the Ltac language, which automate repetitive tasks across proofs. Often, Coq libraries offer a set of tactics to abstract away technical details and perform complex actions. ViCAR aims to provide such tactics for users verifying projects which have categorical structure.

**Symmetric monoidal categories** Following the definition of Selinger [19], a (planar) monoidal category consists of a base category $\mathscr{C}$ equipped with a bifunctor $\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ called the tensor product. The tensor product is required to be unital and associative, which means there are natural isomorphisms whose components are, for objects $A, B, C \in \mathscr{C}$,

$$\lambda_A : 1 \otimes A \xrightarrow{\sim} A, \qquad \text{(left unitor)}$$

$$\rho_A : A \otimes 1 \xrightarrow{\sim} A, \qquad \text{(right unitor)}$$

$$\alpha_{A,B,C} : (A \otimes B) \otimes C \xrightarrow{\sim} A \otimes (B \otimes C). \qquad \text{(associator)}$$

$$\begin{array}{ccc} (A \otimes 1) \otimes B & \xrightarrow{\alpha_{A,1,B}} & A \otimes (1 \otimes B) \\ & {\scriptstyle \rho_A \otimes \mathrm{id}_B} \searrow \quad \swarrow {\scriptstyle \mathrm{id}_A \otimes \lambda_B} & \\ & A \otimes B & \end{array}$$

$$\begin{array}{ccccc} & & (A \otimes B) \otimes (C \otimes D) & & \\ & \nearrow {\scriptstyle \alpha_{A \otimes B,C,D}} & & \searrow {\scriptstyle \alpha_{A,B,C \otimes D}} & \\ ((A \otimes B) \otimes C) \otimes D & & & & A \otimes (B \otimes (C \otimes D)) \\ \downarrow {\scriptstyle \alpha_{A,B,C} \otimes \mathrm{id}_D} & & & & \uparrow {\scriptstyle \mathrm{id}_A \otimes \alpha_{B,C,D}} \\ (A \otimes (B \otimes C)) \otimes D & & \xrightarrow{\alpha_{A,B \otimes C,D}} & & A \otimes ((B \otimes C) \otimes D) \end{array}$$

Figure 1: The triangle identity (left) and the pentagon identity (right)

These natural isomorphisms are required to be coherent. Concretely, this means any diagram of a certain formal class made only of associators and unitors must commute (for details, see Selinger [19]). This requirement boils down to proving the commutativity of just two types of diagrams [9]. Specifically, a monoidal category $\mathscr{C}$ is coherent if and only if the diagrams in Figure 1 commute for all $A,B,C,D \in \mathscr{C}$ [9]

A monoidal category can further be braided, meaning that there is a natural isomorphism whose components are, for objects $A,B \in \mathscr{C}$,

$$\beta_{A,B} : A \otimes B \xrightarrow{\sim} B \otimes A. \qquad \text{(braiding)}$$

Again, two diagrams called the hexagon identities, which this time involve braiding, must commute. Finally, a braided monoidal category is symmetric if, for all $A,B \in \mathscr{C}$,

$$\beta_{A,B} \circ \beta_{B,A} \simeq 1. \qquad \text{(symmetry)}$$

Essentially, symmetric monoidal categories have an almost-commutative tensor product. We choose to focus on monoidal and symmetric monoidal categories because of their prevalence and natural correspondence to string diagrams [19]. String diagrams are a graphical way to represent morphisms, useful for parsing complex structure. One of our goals is to unify verification, which is almost always text-based, with visual reasoning. String diagrams are the platform for doing so for monoidal categories. We define them concretely in Section 4.

**Categories in verification** In mathematics and computer science, symmetric monoidal categories are everywhere. Commonly seen examples include the category of sets, the category of finite vector spaces, and the simply-typed lambda calculus. In verification, active projects whose core structure is a symmetric monoidal category include the verification of the ZX-calculus [16] and that of causal separation diagrams [3]. The ZX-calculus is a graphical language for expressing quantum computation, while causal separation diagram allow us to reason about parallel processes. Both constructs independently satisfy the definitions of a symmetric monoidal category, though this fact is not directly used in their verification.

Despite being in completely different domains, because of their shared structure, the ZX-calculus and causal separation diagram have properties in common that should be exploited to ease their verification. This is the idea that inspired ViCAR and the gap in formal verification that we wanted to address[1]. We bridge this gap with our framework for instantiating monoidal categories, our generalized rewriting tactics, and our morphism visualizer. We prioritize automation and ease—we want users to be able to use ViCAR's features in their own proofs with minimal additional effort.

Other projects attempt to unify categorical reasoning, proof assistants, and visualization. The Chyp proof assistant, for one, allows users to state rewrite rules axiomatically and produces string diagrams to visualize the rules in action [15]. ViCAR takes the alternative approach of requiring users to prove their structures are instances of predefined category typeclasses. In exchange for this effort, ViCAR proofs can be used within the greater context of the Coq proof assistant and augment existing Coq projects. ViCAR's approach allows us to use categorical reasoning to verify existing software, while Chyp is able to more easily handle rewrites modulo associativity.

# 3 Constructively defining categories in Coq

There are several preexisting examples of implementing category theory in Coq [10, 12]. Though many of these libraries have significant developments, we found they did not align with all of our goals. For

[1]We note that VyZX and causal separation diagrams are implemented in different proof assistants, Coq and Agda, respectively. We chose to use Coq for ViCAR.

```
Class MonoidalCategoryCoherence {C : Type} {cC : Category C}
  {cCh : CategoryCoherence cC} (mC : MonoidalCategory cC) : Type := {
    triangle (A B : C) :
        α_ A, I, B ∘ (id_ A ⊗ λ_ B)
        ≃ ρ_ A ⊗ id_ B;
    pentagon (A B M N : C) :
        (α_ A, B, M ⊗ id_ N) ∘ α_ A, (B × M), N ∘ (id_ A ⊗ α_ B, M, N)
        ≃ α_ (A × B), M, N ∘ α_ A, B, (M × N);
    (* Remainder omitted *)
}.
```

Listing 1: The triangle and pentagon identities in the `MonoidalCategoryCoherence` typeclass.

instance, we want to separate structural definitions from coherence conditions to enable easy visualization of category instances, independent of their semantics. Moreover, when working with categories in practice, some notion of morphism equivalence is assumed, so we benefit by making this explicit in formal verification. For ViCAR, we are interested in category theory as a means to generalize shared structure. Our library is to be instantiated by active verification projects across a range of domains. We made a number of technical decisions to reflect this purpose.

We implement our categorical definitions using a hierarchy of Coq typeclasses [20], a mechanism similar to interfaces in object-oriented programming. Typeclasses specify and label a collection of types, possibly dependent on each other, and instances of that typeclass must provide a concrete term for each type. For example, Listing 1 gives the part of the typeclass for monoidal categories which translates the coherence conditions from Figure 1. Typeclasses can also inherit from each other. ViCAR's typeclass hierarchy starts with the base category, then monoidal category, braided monoidal, and finally symmetric monoidal. A project may instantiate as many typeclasses as is suitable for its purposes.

A benefit of using typeclasses is Coq's inference mechanism, which automatically searches for typeclass instances [20]. This avoids having to reference a particular instance every time one of its terms is used. For example, suppose an instance `catC` of our category typeclass has been declared whose objects have type $C$. Then, if $A,B$ and $M$ are terms of type $C$ and `f` and `g` have types $A \rightsquigarrow B$ and $B \rightsquigarrow M$, the expression `f ∘ g` would type check without the user having to explicitly point to the particular instance. Coq would automatically retrieve it, determining that morphism composition $\circ$ should be taken as defined by `catC`.

One important way ViCAR differs from existing formalizations of category theory is its separation of structural definitions from coherence conditions. Each abstract category has a typeclass containing just the necessary structures, such as identity or the associator, and a different typeclass specifying its coherence conditions. There are two separate hierarchies accordingly. The benefit is that users can access the visualizer without having to prove coherence, which may be more demanding. Instantiating a structural typeclass with existing definitions is enough to begin using our visualizer. Of course, without proving coherence, one has no guarantee that the provided structure actually satisfies the criteria of an abstract category, and therefore cannot use our automation.

Another way ViCAR diverges from other formalizations is by requiring users to supply an explicit equivalence relation for morphisms, denoted $\simeq$, instead of always using built-in equality. Using such an equivalence relation is necessary for many standard constructions because Coq does not support performing a quotient by an equivalence relation. By working over the supplied morphism equivalence, we maintain this flexibility that many implementations rely on. We also chose to use a diagrammatic compose for the notation $\circ$, as our project focuses on visualization.

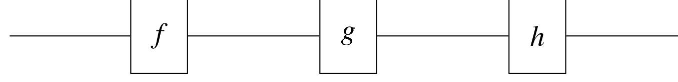
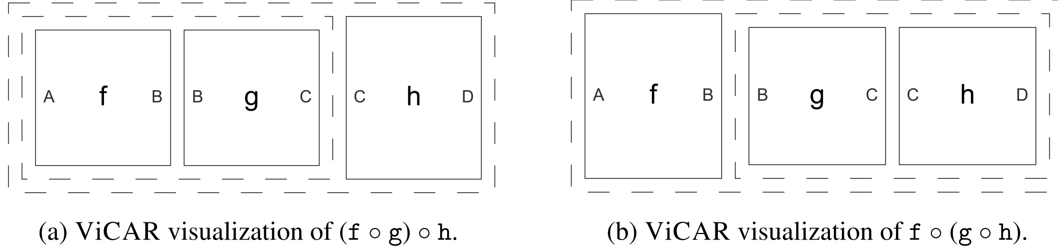



Figure 2: Visualization of both $f \circ (g \circ h)$ and $(f \circ g) \circ h$ in traditional string diagrams.




(a) ViCAR visualization of (`f` ∘ `g`) ∘ `h`.

(b) ViCAR visualization of `f` ∘ (`g` ∘ `h`).

Figure 3: How ViCAR renders parentheses.

# 4 Visualization

**Categorical String Diagrams** Reasoning about morphisms in categories is assisted by the use of string diagrams, an associated graphical language. String diagrams visually represent monoidal categories that are ordinarily represented in text form. Their notation focuses on morphisms rather than objects, and can therefore very concisely represent complicated expressions. We visualize our categories as string diagrams to reduce cognitive overhead during proof.

Traditional string diagrams omit details such as associativity, so both $f \circ (g \circ h)$ and $(f \circ g) \circ h$ are visualized as Figure 2, for example. This is useful for pen-and-paper proofs but not for proof assistants—formal proofs require each "obvious" detail to be addressed. In Coq, $f \circ (g \circ h)$ and $(f \circ g) \circ h$ are two distinct terms, equivalent only via rewriting (applying a theorem which states they are equal). Identifying necessary rewrites is essential to the proof engineering workflow, so we tweaked ViCAR's string diagram notation to maintain all parenthesizing explicitly. We represent these parentheses by circumscribing boxes, and the differences can be easily identified as in Figure 3.

Our visualizations, while convenient for formal proof, are more verbose and add layers of complexity over the simpler diagrams. Future extensions of ViCAR hope to solve this problem, by automatically handling structure. This would allow for diagrams to be canonically represented and for proof engineers to no longer focus on structural rewrites. While work has been done on rewriting in Coq modulo associativity and commutativity, we found none of it sufficient for what ViCAR needs [2]. The current gap between existing modulo associativity rules and this project relates to rules that depend on the interactions between two operators, like $(f \circ g) \otimes (h \circ t) \simeq (f \otimes h) \circ (g \otimes t)$. There are promising directions being developed on top of e-graph based equality saturation, as we discuss in Section 7.

**Visualization semantics and workflow** ViCAR allows for visualization of morphisms and morphism equivalences over base category instances up to symmetric monoidal ones. We unlock functionality as we deal with more expressive categories.

- A morphism $f : A \rightsquigarrow B$ is visualized as a quadrilateral, marked with $A$ on the left and $B$ on the right.
- The identity morphism for $A$ is denoted by a wire (horizontal line) with $A$ annotating the input and output positions.
- Morphisms are composed by placing them side-by-side horizontally.

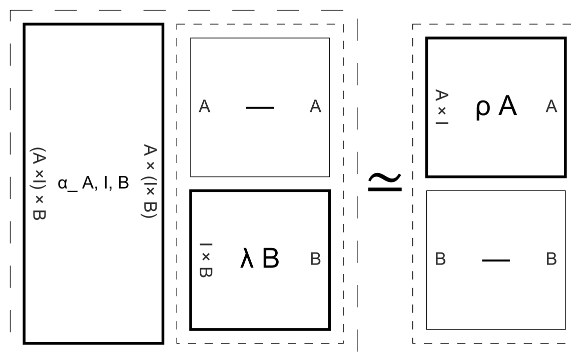



Figure 4: A visualization of the triangle identity.

- In monoidal categories, morphisms are tensored by placing them side-by-side vertically.
- Taking the inverse of the morphism is rendered in a box attached to left of the morphism.
- An isomorphism is a morphism with an emphasized bounding box.
- Category-specific terms, such as the associator, are identified as such (with the notation from Section 2).
- In braided monoidal categories, the braiding is rendered as a big cross.

An example which uses several of these features can be seen in Figure 4, which displays the statement of the triangle identity from Listing 1.

One of ViCAR's most convenient features is its integration into the Coq proof-writing workflow. ViCAR is connected to the coq-lsp VSCode extension [8], which type checks dynamically and prints the current goal state (hypotheses and statements yet to be proved) based on cursor position within a proof. The visualizer renders a string-diagrammatic goal state, alongside the printed one, and updates automatically as the goal changes. The user's setup can be seen in Figure 5.

Because of this integration, the visualizer is able to use real-time hypothesis information to inform rendering choices. This is useful for distinguishing overloaded notation or opaque variables. For example, if the variable `f` in a proof state is a morphism from `A` to `B`, the visualizer will make this clear while the goal may not show it explicitly. Association and the effect of braiding may also be unclear from the proof state. The visualizer consults the hypothesis to check the type of each morphism and uses this information to label the inputs and outputs of each morphism's box.

# 5 Automation

Categorical structure gives a well-defined domain on which we can develop partial proof automation. Specifically, we can automate proofs of statements that two morphisms are equivalent. This is the form of many theorems in verification projects with monoidal structure. For example, theorems about matrices take the form that certain matrices are equal, and theorems about the ZX-calculus take the form that diagrams are proportional. These types of proofs generally consist of rewrites and share common techniques, such as manipulation of the underlying categorical structure. In this section, we explain ViCAR's tactics and how they can help with these proofs.

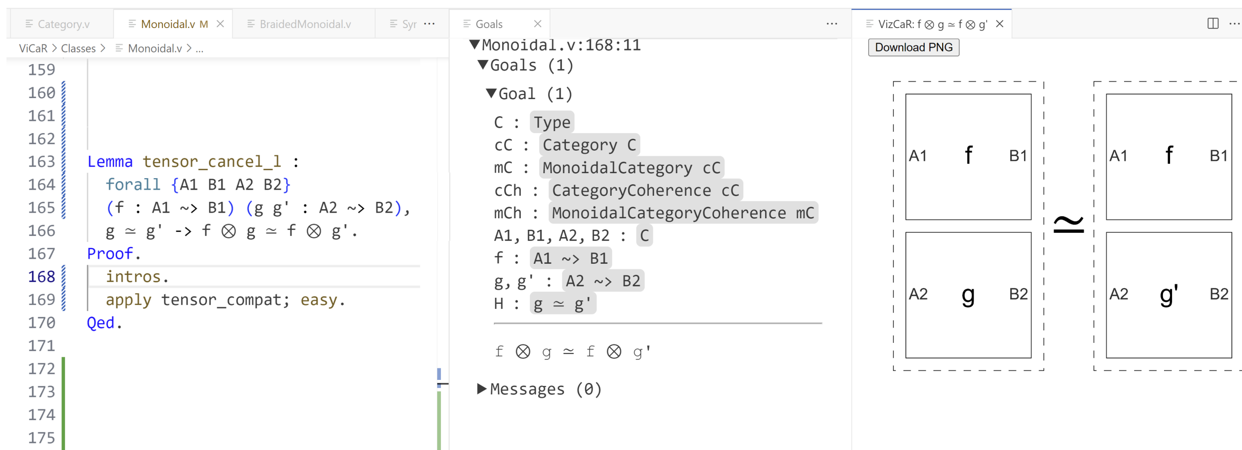


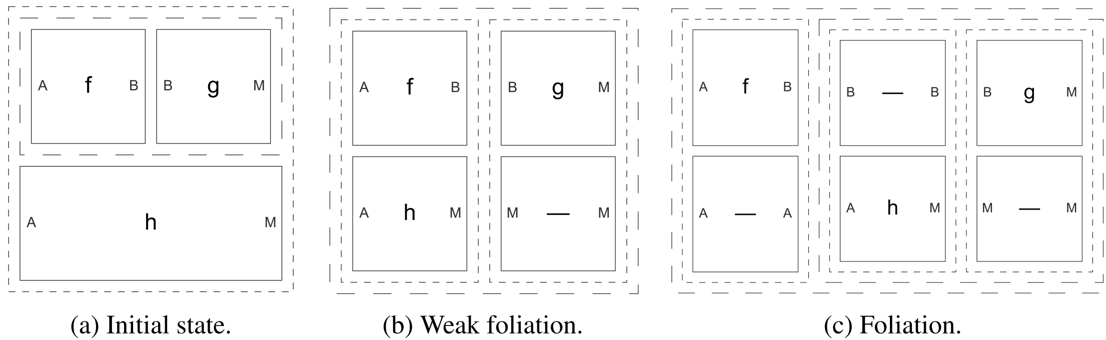

Figure 5: User's proof-writing IDE state

**Coercing to categories** Coq will often fail to apply general categorical lemmas to proofs about specific typeclass instances because it cannot recognize which terms correspond to categorical structures. For example, suppose the category of matrices (defined in full in Section 6.3) has been declared as a typeclass instance, with composition and identity given by matrix multiplication and the identity matrix. It would not be possible to rewrite the term `I_n` $\times$ `A` with the lemma whose statement is `id_ A` $\circ$ `f` $\simeq$ `f`. Coq fails to recognize that $\times$ is the composition of a declared category instance and `I_n` is the identity.

To address this gap, we provide the tactic `categorify`. It performs setup necessary for our automation to identify the structure of the goal. For the category of matrices, calling `categorify` would replace each instance `A` $\times$ `B` of matrix multiplication with categorical composition, `A` $\circ$ `B`, and the same for all other terms used to define to the instance. In practice, when using ViCAR, this serves as a setup tactic. At the beginning of the proof, `categorify` is called, which unlocks all the functionality of the visualizer and automation, provided the relevant structural and coherence instances have been defined.

**Foliation** A common representation of a morphism is a foliation [6]. A foliation is a composition of "stacks," each of which is the tensor of identity morphisms and a single non-identity morphism. Such a representation exists for any diagram and can be considered a standard form and useful for proofs [6]. Formally proving this result for categorical instances, however, requires the complex notion that some morphisms are atomic with respect to this decomposition. For example, in the category of matrices, it is hard to define when a matrix should be decomposable with respect to matrix multiplication and Kronecker product (the tensor product). Our tactic `foliate` computes a foliation of a given morphism from any monoidal categorical instance, proves they are equivalent, and replaces the morphism with its foliation. Often, a full foliation is undesirable, and a more concise form is preferable. The `weak_foliate` tactic performs a partial foliation that allows multiple non-identity morphisms in a stack, but still ensures no stack contains a composition. An example of `weak_foliate` and `foliate` is given in Figure 6 on (`f` $\circ$ `g`) $\otimes$ `h`. Its partial foliation is `f` $\otimes$ `h` $\circ$ `g` $\otimes$ `id_ M` and its full foliation is `f` $\otimes$ `id_ A` $\circ$ (`id_ B` $\otimes$ `h` $\circ$ `g` $\otimes$ `id_ M`).

**Associativity: partnering and rewriting** A recurring task in many Coq projects is dealing with associativity. Coq expressions have explicit association, so to rewrite a subterm of a sequence of compositions, it is often necessary to perform several rewrites using associativity rules. Even the expression `g` $\circ$ `f` $\circ$ `f^-1` cannot be rewritten to `g` directly, requiring first a rewrite using the associativity condition. In larger expressions,




(a) Initial state. (b) Weak foliation. (c) Foliation.

Figure 6: Visualization of foliation tactics on $(\mathtt{f} \circ \mathtt{g}) \otimes \mathtt{h}$.

the reassociations which must be performed can be quite laborious to specify manually. We provide several tactics to automate this work.

First, we define the `partner` tactic, which takes two terms as arguments and attempts to reassociate the goal to make these terms syntactically adjacent. After calling `partner f g`, we get $(\mathtt{f} \circ \mathtt{g})$ as a subterm of the goal, assuming it can isolated by reassociation.

Building on this technique, we also define the `assoc_rw` tactic, which takes a lemma as an argument and attempts to reassociate the goal such that the lemma can be rewritten. This works for any lemma whose conclusion, possibly quantified over arguments, has the form $F \simeq g$, where $F$ is some sequence of compositions and $g$ is any morphism. This tactic allows a user to entirely ignore the association of the goal, and simply rewrite according to the sequence of morphisms present in the goal. For example, given a lemma $\mathtt{f} \circ \mathtt{g} \simeq \mathtt{h}$, `assoc_rw` would rewrite its occurrence in the expression $\mathtt{i} \otimes (\mathtt{e} \circ \mathtt{f} \circ \mathtt{g})$, even though this requires reassociation within the argument to the tensor product.

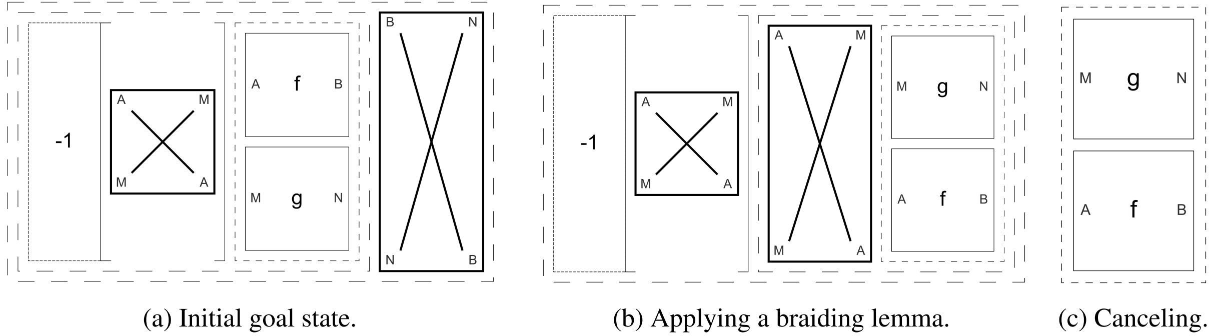
These tactics together entirely obviate the need for manual association of the goal. This delivers on one part of the motivation for using string diagram representations of morphisms: such representations implicitly encode the coherence conditions of monoidal categories by means of topological irrelevance [19]. While full topological irrelevance is hard both to define and to prove within Coq, suppressing the consideration of association is a first step in this direction.

**Simplifying the goal state** Building on these tactics, we provide a number of tactics to simplify the goal state. The tactic `cancel_isos` will cancel any isomorphism adjacent to its inverse, independent of associativity. This is particularly useful for eliminating structural morphisms that accumulate by applications of naturality properties. The tactic `cat_simpl` combines this cancellation with the removal of identity morphisms. These tactics give a quick way to perform common simplifications of a goal without having to consider association. The tactic `cat_easy` attempts to solve goals that are fundamentally structural, and not too complicated. It will cancel isomorphisms and identities, right-associate the goal, and perform weak foliation until the goal is solvable by reflexivity.

Listing 2 and Figure 7 give an example of these tactics in action to almost completely automate a proof. The `assoc_rw` tactic automatically associates the goal to apply the given lemma, a braiding coherence condition. `cancel_isos` then reassociates and cancels out the braiding and its inverse. These tactics can be applied to any proof for a concrete instance, after first calling `categorify`.

```
Lemma assoc_rw_example {A B M N : Cobj}
  (f : A ~> B) (g : M ~> N) :
  (β_ A, M)^-1 ∘ f ⊗ g ∘ β_ B, N
  ≃ g ⊗ f.
Proof.
  assoc_rw braiding_natural. (* apply braiding lemma *)
  cancel_isos. (* cancellation *)
  reflexivity.
Qed.
```

Listing 2: ViCAR's tactics being used to easily solve an example lemma.




(a) Initial goal state. (b) Applying a braiding lemma. (c) Canceling.

Figure 7: Visualization at each step of Listing 2.

# 6 Example uses of ViCAR

## 6.1 *VyZX*

The ZX-calculus is a graphical language to represent quantum operations [5]. Each ZX-diagram represents a linear transformation between qubits and consists of red and green nodes connected by wires. ZX's appeal is that its rewrite rules are easily visualized as graph manipulations. ZX-diagrams can be visually transformed into any equivalent diagram using a finite set of rewrites. The ZX-calculus has been used for simulation [14], circuit optimization [13], and fault tolerance [1] work. For a deeper introduction to the ZX-calculus, refer to one of these surveys [21, 4].

*VyZX* (Verifying the ZX-Calculus) is an effort to formalize the ZX-calculus in Coq [16]. It gives inductive definitions for diagrams and interprets them through standard matrix semantics for the ZX-calculus. The ZX-calculus corresponds to a symmetric monoidal category whose objects are natural numbers (representing the number of input and output wires) and whose morphisms are ZX-diagrams. Horizontal composition of morphisms (connecting the input and output wires of two diagrams) is associative. The tensor product is given by vertically stacking diagrams, corresponding to addition on natural numbers with identity object 0. The morphism identity given for a natural number $n$ is $n$ wires with no nodes, since composition with this object does not affect any diagram.

As *VyZX* is a well-developed verification library, most of these categorical structures and necessary lemmas already existed. As a result, the coherence conditions for these morphisms were straightforward to prove. The braiding was easily implemented, but proving naturality was a significant task. Fortunately, this task was made easier by proving naturality of braiding for matrices, as discussed in 6.3.

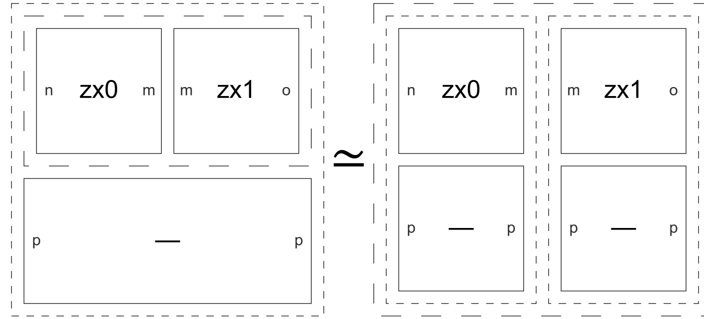



Figure 8: Visualization of the statement (zx0 ↔ zx1) ↕ n_wire p ∝ (zx0 ↕ n_wire p) ↔ (zx1 ↕ n_wire p).

The VyZX visualizer, ZXVIZ, displays ZX-diagrams by parsing an expression into its building blocks and generating a string diagram. We modified ZXVIZ to work for general category instances, abstracting the specific display style used for ZX-diagrams and significantly reworking the foundation of the visualizer to work in the context of a general category. Figure 8 gives an example of ViCAR's visualizer for a ZX-specific lemma, where zx0 and zx1 are ZX-diagrams. Its string diagram makes clear this lemma is really just a structural result.

ViCAR grew out of *Vy*ZX in an attempt to separate rules specific to the ZX-calculus from rules which are common to any symmetric monoidal category. Typical ZX-calculus literature does not reason about such structure but rather about ZX-diagrams as graphs. Therefore, future *Vy*ZX development can benefit from ViCAR's automation by being able to better ignore structural manipulation of ZX-diagrams.

### 6.2 Calculus of Relations

In contrast to the previous implementation, the calculus of relations is not a pre-existing Coq project. Its implementation follows the book *Picturing Quantum Processes* [6]. To adapt this work to Coq, we define the objects of our category to be types, and we define morphisms between types T and S to have type T → S → Prop. As this is defined over any T and S, it can be easily applied to any specific example. By defining base relations such as sibling and parent over a type person, we can construct and visualize relations such as uncle in Figure 9. While relations are a toy example, they do demonstrate some insights.

Our implementation of relations was made with ViCAR in mind, and so all proofs are intended for our typeclasses. However, categorical structure is far from a complete description of the properties of relations. For example, it is natural to want to describe the transitive closure of a relation R : T → T → Prop. There is no clear way to do this solely using categorical construction. Instead, we are able to use Coq's built-in capabilities to describe a very similar construct: given R : T → T → Prop, we define R^n, the repeated application of R $n$ times. We can then construct the transitive closure diagrammatically as ρ_ A ∘ (id_ A ⊗ any_nat) ∘ R^n, where any_nat is a relation that relates unit to any nat. By having ViCAR embedded in an existing proof assistant, we gain the ability to easily create new morphisms as needed and can reason about both structure and properties specific to a particular instance, together.

### 6.3 Matrices

The collection of vector spaces over a field $k$ form the category $\mathrm{Vect}_k$ whose morphisms are linear transformations of spaces. This is a monoidal category with respect to the tensor product of vector spaces, with identity

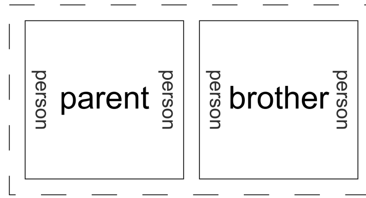



Figure 9: Visualizing the "uncle" relation between two people, `parent` $\circ$ `brother`. (Note that this is diagrammatic composition; parent precedes brother.)

element $k$. Moreover, this monoidal category is braided, as $V \otimes W \cong W \otimes V$.

A commonly-used subcategory of $\text{Vect}_k$ is the category $\text{FinDimVect}_k$ of finite-dimensional vector spaces, which is a braided monoidal category in exactly the same way. $\text{FinDimVect}_k$ has a skeleton given by vector spaces of the form $k^n$ for $n \in \mathbb{N}$. This skeleton is isomorphic to the category whose objects are natural numbers and whose morphisms from $n$ to $m$ are the $n \times m$ matrices over $k$, representing linear transformations $k^n \to k^m$. We implement that category for $k = \mathbb{C}$ based on the matrices in the `QuantumLib` library [11], which provides verified mathematics for quantum computing in Coq. *VyZX* defines its semantics in terms of `QuantumLib` matrices.

`QuantumLib` already implements most of the definitions and lemmas required to instantiate a category. Matrices in `QuantumLib` are defined as functions $\mathbb{N} \to \mathbb{N} \to \mathbb{C}$. The dimensions of `QuantumLib` matrices are not enforced by the type checker, but instead given as "phantom types" to guide the programmer [18]. This lack of strict dimensions makes the construction and use of the associator and unitors very straightforward.

We implement our category instances using bounded equivalence ($\equiv$) as our morphism equivalence, which says that matrices $A, B$ of dimension $n \times m$ are equivalent if they are equal on all entries within their bounds. The majority of the work in showing matrices are a braided monoidal category involves showing the naturality condition of the braiding. In this category, the braiding is given by the commutation matrices $K_{n,m}$, which have the property that for any $n \times m$ matrix $A$ and $p \times q$ matrix $B$, we have $K_{p,n}(A \otimes B) = (B \otimes A)K_{q,m}$ [17]. Due to phantom types, the associator and unitor can be defined as the identity matrix interpreted with the appropriate dimensions. In proofs, these matrices can be immediately canceled, which makes the coherence conditions easy to prove.

A diagrammatic representation of matrix expressions involving the Kronecker can reveal facts which textual representations obscure. For instance, Figure 10 shows the distributivity of the Kronecker product over matrix multiplication. The diagrammatic depiction of this equality highlights the structure of the vector spaces on which these matrices act, making a non-obvious equation more believable.

# 7 Future Directions

ViCAR provides a framework for visualization and automated rewriting by defining typeclasses for categorical and monoidal structure. The language of string diagrams can further describe rigid symmetric monoidal categories, with the unit and counit of the category depicted as half turns [19]. ViCAR can be extended with this structure, allowing it to capture a wide variety of process theories.

The coherence conditions in our typeclass definitions have been proven sufficient to show the coherence of the respective categorical structures. If formalized in Coq, these results could enable much stronger diagrammatic reasoning by allowing large purely structural rewrites to be performed immediately, rather than having to manually prove the specific instance of coherence necessary to perform the replacement.

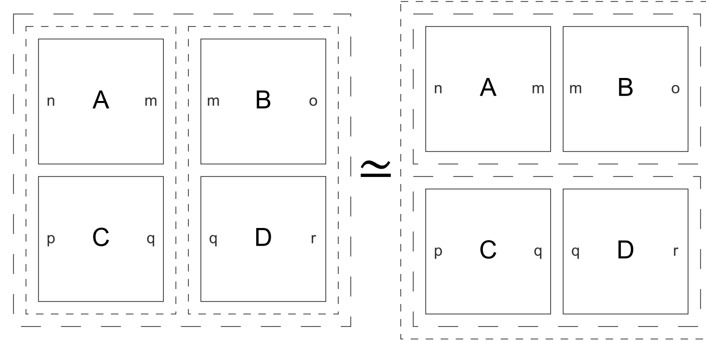



Figure 10: Visualization of the Kronecker mixed-product, $(A \otimes C) \times (B \otimes D) \equiv (A \times B) \otimes (C \times D)$.

Many symmetric monoidal categories are interpreted using a semantics function. In fact, the semantics are often used to define morphism equivalence. We aim to provide infrastructure within ViCAR to allow translation between different symmetric monoidal categories for semantic interpretation. This would greatly simplify constructs in categories equipped with a ViCAR-based semantics by lifting them.

Seeing the advantage that the `assoc_rw` tactic gives us, we want to further expand the capabilities of rewriting structure. In fact, we would like to completely be able to rewrite without worrying about structure. To achieve this goal, we are working on an e-graph equality saturation-based solver for different structural configurations. The solver will work by ingesting Coq statements into a custom AST and then the e-graph solver will use the structural rewrite rules in ViCAR. Once a proof is found, it will be exported into Coq and checked.

## References


[1] Hector Bombin, Daniel Litinski, Naomi Nickerson, Fernando Pastawski & Sam Roberts (2023): *Unifying flavors of fault tolerance with the ZX calculus*. arXiv:2303.08829.

[2] Thomas Braibant & Damien Pous (2011): *Tactics for reasoning modulo AC in Coq*. In: *International Conference on Certified Programs and Proofs*, Springer, pp. 167–182.

[3] Jonathan Castello, Patrick Redmond & Lindsey Kuper (2023): *Inductive diagrams for causal reasoning*. arXiv:2307.10484.

[4] Bob Coecke (2023): *Basic ZX-calculus for students and professionals*. arXiv:2303.03163.

[5] Bob Coecke & Ross Duncan (2008): *Interacting Quantum Observables*. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir & Igor Walukiewicz, editors: *Automata, Languages and Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 298–310, doi:10.1007/978-3-540-70583-3_25.

[6] Bob Coecke & Aleks Kissinger (2017): *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, doi:10.1017/9781316219317.

[7] The Coq Development Team (2012): *The Coq Reference Manual, version 8.4*. Available electronically at `http://coq.inria.fr/doc`.

[8] The Coq LSP Developers (2023): *GitHub - ejgallego/coq-lsp: Visual Studio Code Extension and Language Server Protocol for Coq — github.com*. `https://github.com/ejgallego/coq-lsp`.

[9] Samuel Eilenberg & G Max Kelly (1966): *Closed categories*. In: *Proceedings of the Conference on Categorical Algebra: La Jolla 1965*, Springer, pp. 421–562.

[10] Jason Gross, Adam Chlipala & David I Spivak (2014): *Experience implementing a performant category-theory library in Coq*. In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 5*, Springer, pp. 275–291.

[11] INQWIRE Developers (2022): *INQWIRE QuantumLib*. Available at `https://github.com/inQWIRE/QuantumLib`.

[12] John Wiegley (2022): *Category Theory in Coq*. Available at `https://github.com/jwiegley/category-theory`.

[13] Aleks Kissinger & John van de Wetering (2020): *PyZX: Large Scale Automated Diagrammatic Reasoning*. In Bob Coecke & Matthew Leifer, editors: Proceedings 16th International Conference on *Quantum Physics and Logic*, Chapman University, Orange, CA, USA., 10-14 June 2019, *Electronic Proceedings in Theoretical Computer Science* 318, Open Publishing Association, pp. 229–241, doi:10.4204/EPTCS.318.14.

[14] Aleks Kissinger & John van de Wetering (2022): *Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions*. *Quantum Science and Technology* 7(4), p. 044001.

[15] Alex Kissinger (2023): *GitHub - akissinger/chyp: An interactive theorem prover for string diagrams — github.com*. `https://github.com/akissinger/chyp`.

[16] Adrian Lehmann, Ben Caldwell, Bhakti Shah & Robert Rand (2023): *VyZX: Formal Verification of a Graphical Quantum Language*. arXiv:2311.11571.

[17] Jan R. Magnus & H. Neudecker (1979): *The Commutation Matrix: Some Properties and Applications*. *The Annals of Statistics* 7(2), pp. 381–394. Available at `http://www.jstor.org/stable/2958818`.

[18] Robert Rand, Jennifer Paykin & Steve Zdancewic (2018): *Phantom Types for Quantum Programs*. Available at `https://popl18.sigplan.org/event/coqpl-2018-phantom-types-for-quantum-programs`. Talk at The Fourth International Workshop on Coq for Programming Languages (CoqPL '18).

[19] P. Selinger (2010): *A Survey of Graphical Languages for Monoidal Categories*. In: *New Structures for Physics*, Springer Berlin Heidelberg, pp. 289–355, doi:10.1007/978-3-642-12821-9_4.

[20] Matthieu Sozeau & Nicolas Oury (2008): *First-Class Type Classes*. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, Springer-Verlag, Berlin, Heidelberg, p. 278–293, doi:10.1007/978-3-540-71067-7_23. Available at `https://doi.org/10.1007/978-3-540-71067-7_23`.

[21] John van de Wetering (2020): *ZX-calculus for the working quantum computer scientist*. arXiv:2012.13966.